Technical Report
COMP1100 Assignment 3

Jacob Bos
ANU u7469354

May 22, 2022

Lab: Tuesday 11am

Tutor: Abhaas Goyal

Word-count beyond cover page at ≤ 1500 words

Contents

1 Introduction

The program detailed herein is an implementation of a few AI's for solving the game Fanorona with complimentary unit tests.

2 Documentation

2.1 Design Documentation and Technical Decisions

First capture move is little more complex than the provided `firstLegalMove`, it is "content" with taking the head of the list of possible capturing moves as provided by the function `captures` else returning the first legal move. This was used to test the greedy AI, because a greedy should on average perform better.

The Greedy AI has a simple functionality. The main function `greedy` cases on which player's turn it is given the provided gameState and chooses whether to maximize (Player1) or minimize (Player2) the heuristic value and calls `greedyHelp` with the appropriate evaluator to output a pair containing the ideal move and its heuristic value. It does this by calling `greedyHelp` which recurses through a list of moves and their values and either minimizes or maximizes it. This list is a mapping of the list provided by `legalMoves` to a list of pairs of moves of each move with the value of the move created by the function `diffPieces` applied to the `applyMove` of the move and initial state.

Minimax uses two recursive tree structures, the first `GameTree`, stores all the possible gamestates and is generated through an infinite recursion in `gameTree` which takes a state, puts it into a node and then maps `gameTree` to all its children states which are generated through a mapping of `applyMove` to a list of `legalMoves` which is then recursively purged of its `[Maybe GameState]` type by `purge` to become `[GameState]`.

The second tree structure, `evalTree` is the same as `GTree` except that it contains a value on each node corresponding tpo the best possible outcome (heuristic value) for the player who's state is at that node and is pruned to a given *move depth.* The `evalTree` is generated by recursively by `pruneMinMax` which cases firstly on integer depth given to the function, if the depth is zero then it evaluated the heuristic value at that node and then terminates that branch. If not, it then cases on the state held in the node, if the state contains a `GameOver` turn then it does as if the depth was zero, terminating the tree. If the state at a node contains `Turn Player1` then it assigns the maximum of the values in its child `EvalTree` nodes else if the turn is the `Player2`, the minimizing player, it assigns the minimum value of its child nodes to the given node. This results in the best possible outcome for the player in the initial state ending up in the head node.

The heuristic value used is the difference in number of pieces between `Player1` and `Player2` and is calculated by `heuristicVal` which takes the pair output of the provided `countPieces` function and then takes the difference in the number of pieces.

To then retrieve the best move we note that the best move is at the same depth in the `legalMoves` list as best value stored in the head node is in the list of child `evalTree` nodes. This is because the list of children nodes is produced by a mapping on the `legalMoves` list. Consequently to find the best move we extract the value from the head of the output of `pruneMinMax` using `getVal` and then find it's depth recursively in the list provided by mapping `getVal` to the child nodes at depth 1 using `findDepth`. Consequently the function `getMove` uses the (!!) operator to extract the best move at its expected depth in the `legalMoves` list.

The other AI's

2.2 Program Design / Structure

2.3 Assumptions

3 Testing

Unit tests

Performance tests

4 Reflection

4.1 Design Choices

4.2 Reflection